Extreme Programming

Rapid Development for Web-Based Applications

Frank Maurer and Sebastien Martel • *University of Calgary*

As software organizations continue to move toward Web-based systems development, they often assign or outsource such projects to small teams of highly qualified, but often relatively young, developers. Frequently, the developers' attitude is less than positive toward software engineering practices—particularly software process improvement initiatives and metrics collection.

Part of the problem is the business context: Web-based applications demand faster time-to-market and the continual integration of new requirements. Such demands have increased the popularity of agile software processes, which let teams increase development productivity while maintaining software quality and flexibility.

Agile processes like extreme programming (XP),[1-3] Scrum,[4] Crystal,[5] and adaptive software development[6] aim to increase a software organization's responsiveness while decreasing development overhead. They focus on delivering executable code and see people as the strongest ingredient of software development. Here, we offer an overview of the philosophy and practice behind XP, which is currently the most popular agile methodology.

XP Overview

XP consists of 12 related practices and works best for small teams of 5 to 15 developers. Rather than focus on paper-based requirements and design documentation, XP concentrates on producing executable code and automated test drivers. This focus on source code makes XP controversial, leading some to compare it to hacking. We believe this comparison is unjustified because XP highly values simple design, and counters hacking claims by emphasizing refactoring, strong regression testing, and continuous code inspections through pair programming.

XP and Web Development

XP's focus on small teams lets it replace paper-based documentation with face-to-face communication. Hence, it's a good fit for many Web-based software projects, which often postpone documentation efforts because of time-to-market constraints.

In XP, all developers work closely together so they can communicate informally rather than spending time documenting designs and decisions. As long as teams remain small, this approach pays off: It's faster to talk directly than to write down development knowledge. In addition, direct communication is typically limited to existing issues. To produce documentation, writers often have to make assumptions about what information will be useful for readers. If these assumptions are wrong or if the software design changes drastically, the documentation effort is wasted. As the development organization grows, however, time spent exchanging product knowledge and training new people increases and often renders XP unsuitable.

Productivity Gains

XP's focus on reduced documentation should obviously improve productivity—at least in the short run. In fact, our case study showed strong productivity gains after a switch from a document-

 1089-7801/02/$17.00

centric development process to XP.[7]

We studied a small company with nine full-time programmers developing a Web-based system over 16 months. Aside from the development process, which changed from a fairly traditional object-oriented approach to XP, all aspects of the project stayed constant. We calculated three size metrics by comparing the new release with the previous system version:

- *NLOC* measures new lines of code (Java source code plus HTML code);
- *#methods* measures the number of new methods; and
- *#classes* measures the number of new classes.

Table 1 summarizes our findings and shows productivity gains between 66.3 percent and 302.1 percent based on hard metrics. We measured *effort* as the number of hours billed to the customer. We offer a detailed analysis of our data and results elsewhere.[7]

Table 1. Productivity gains with XP

	NLOC/effort	#methods/effort	#classes/effort
Average Pre-XP	10.2	0.36	0.05
Average XP	17.0	1.45	0.21
Percent Change	66.3	302.1	282.6

XP Practices

XP proposes a set of software development practices to increase productivity while maintaining quality. We group XP's 12 practices according to three key areas: customer satisfaction, software quality, and development process organization.

Customer Satisfaction

Building a high-quality software system is irrelevant if it does not solve the customer's problem. To increase customer satisfaction, XP uses two practices: on-site customer and small releases.

On-site customer. Determining and prioritizing requirements is essential for any successful software project. However, trying to "get the requirements right" before the software is designed and implemented is problematic in Web-based systems, where requirements frequently change. In XP, developers initially document requirements through *user stories*, which are basically textual use-case descriptions (see the "Common XP Terms" sidebar). To clarify these requirements and set priorities, XP uses an on-site customer representative who works with the team.

This practice improves the software's business value: When issues arise, programmers can get customer input immediately rather than speculate on customer preferences. This also lets customers change requirements on very short notice—thereby helping the team flexibly refocus development efforts on the most pressing needs.

Small releases. Given that requirements change often, XP keeps release cycles short and ensures that each release produces a useful software system that generates business value for the customer. Short cycles reduce customer risk, letting the customer quickly terminate projects that fail to deliver business value. A short release cycle also helps developers deal with changing requirements and reduces the impact of planning errors.

Software Quality

XP employs various practices to keep software quality high. Although some might appear unusual, their combined effects ensure that the team maintains high quality without slowing down the development process.

Metaphor. A metaphor represents a coherent view of the system that makes sense to both the business and technical sides and represents "what we are trying to do." The metaphor is sometimes embodied in a single user story that portrays this idea and gives everyone the system basics. In a sense, the metaphor serves as the high-level software architecture.

Testing. Software testing – specifically, automated regression testing—is a key part of XP. The customer defines functional (acceptance) tests, which the development team implements. From a business perspective, these tests verify that the program does what it is supposed to do. According to the XP philosophy, a feature lacking automated tests does not exist.

In XP, programmers write unit tests before they write actual code. Writing test drivers before writing the code forces software developers to think about the problem before programming. The test drivers thus serve as a detailed specification of the method's functionality. This is also one of the intentions of up-front design. The XP team can also use test drivers later to see if the system exhibits the proper functionality after the code is written. They can't do this with paper-based documentation.

Common XP terms

- **User stories:** use cases that briefly capture functional requirements. Developers typically write stories on index cards to describe each system task that the customer desires. Ideally, it should take a team one to five engineering weeks of effort to implement the tasks each story implies. Stories should be testable.
- **Customer:** a role on the development team; the customer chooses which stories the system must satisfy and the order in which to implement them. The customer also defines acceptance tests to verify that the stories function correctly.
- **Unit tests:** tests written from the programmer's perspective to identify possible system malfunctions. The team keeps all unit tests running all the time: Nothing is integrated into the baseline if a unit test fails. If a bug is encountered, the team first writes a unit test that shows the problem, and then fixes it.
- **Functional/acceptance tests:** tests from the customer's perspective that ensure that the system correctly implements a user story.

Simple design. Agile software processes assume that requirements are always changing, the future is uncertain, and the costs of change are not exponential. Hence, the most cost-effective development approach should focus on solving today's problems rather than designing for future changes. In XP, the best software design runs all the test drivers, has no code redundancies, has the fewest possible classes and methods, and is easy to understand. XP does not invest in up-front analysis and design. It trades the potential savings of anticipating change against that of wrongly guessing the system's future direction.

Keeping the software design as simple as possible improves a team's ability to work productively with minimal documentation beyond the source code. When source code is easy to understand, there is no need to document its structure at a higher level of abstraction.

Refactoring. All software deteriorates over time: A once-clean design becomes progressively fuzzier with each change. However, when a software system has minimal documentation, the source code must remain simple and easy to understand. To accomplish this, programmers refactor the code base. That is, they change the code's structure to improve its understandability and maintainability, without changing its functionality.[3] Once a feature is complete, the programmer must ensure that the existing structure is the simplest way to run all the tests. If not, the code must be refactored and simplified.

Automated test drivers ease refactoring efforts by giving programmers feedback after they change the code. If the changes work, they become part of the next baseline. If they don't, the programmer can fix or reverse them right away.

XP uses patterns, but it does so according to a simple idea: Rather than implement with patterns, developers should "refactor to patterns when appropriate and away from patterns when something simpler is discovered."[8] Constant refactoring ensures that the design is always as simple as possible.

Pair programming. In XP, all production code is written by two people working at one machine. One person controls the keyboard and the mouse, and focuses on broad issues such as whether this approach will work and whether it can be simplified further. The other person thinks more strategically and decides if this is the best way to implement the functionality. The pair switches roles frequently throughout the day.

Managers often object to pair programming on the grounds that it doubles a project's programming costs. However, in their controlled experiment measuring pair-programming overhead, Williams and colleagues found that it takes only approximately 15 percent more effort than solo programming, but increases both software quality and programmers' job satisfaction.[9] (In situations where pair programming is unacceptable, teams might consider replacing it with software measurements and inspections.[10])

Project Management

To ease the project-management burden, XP includes practices aimed at reducing management overhead, while keeping the customer's interest at close range.

The planning game. To chart the next release's scope, XP uses the "planning game," that takes both business priorities and development team realities into account. Business interests can be represented by various people, including marketing personnel or a customer's employee. All that matters is that whoever it is has the required knowledge to decide

- *Scope and priority*. What are the required system features? Which features are most important and must be added now? Which features can be postponed?
- *Release date*. When must the next release be available?

Obviously, to make such decisions, businesspeople need information from developers. Among the

technical side's responsibilities are

- *Estimates*. How much effort is required to implement a new feature or fix a bug? How much work can be put into the next release?
- *Consequences*. How will various business decisions impact the development process and development effort?
- *Process*. How will the team work together and be organized?

To estimate effort, XP uses *ideal engineering time* (IET), which measures a task's difficulty or complexity, and *velocity*, which determines how many IET points a team can use within a set time period. The team determines velocity based on past experience. For example, if the last release's team got x IETs done in two weeks, they would set that as the mark for the new release as well.

The planning game's goal is to balance customer interests with the expertise of the development team. The team estimates task effort, while the customer picks the tasks for the next iteration, constrained only by the team's velocity.

Sustainable development. According to XP philosophy, no one can work 60-hour weeks consecutively without affecting product quality. As Kent Beck put it:

> I want to be fresh and eager every morning, and tired and satisfied every night. On Friday, I want to be tired and satisfied enough that I feel good about two days to think about something other than work. Then on Monday I want to come in full of fire and ideas.[1]

Although management or customers can push a team to work long overtime hours, the result is usually reduced quality, burned-out developers, and a high turnover rate.

Collective ownership. In XP, the team members collectively own the code base. Anyone who can add value to any portion of the code at any time is required to do so. This practice succeeds in part because of the automated test suite: Programmers will get feedback on any code they modify to see if it works or not. Having automated test drivers lets programmers modify the code more freely and with less fear of unknown repercussions.

Coding standards. Because developers program different parts of the system with various team members, coding standards are a must. A coding standard makes the code easier to understand and improves consistency among team members. The standard should be easy to follow and adopted voluntarily.

Continuous integration. Developers should integrate code as often as possible, and at least once a day. This ensures that there is always an executable system version available that contains all new features and can serve as the baseline for all work.

A good way to do continuous integration is to dedicate a machine, letting pair programmers take turns integrating and testing their code. If the tests fail, the pair can undo their changes and work on their code until it passes all the tests.

Conclusion

XP's 12 core practices are closely related, and implementing only a few will not necessarily bring all potential benefits. For example, you cannot have people modifying the entire code base without automated test drivers in place. On the other hand, some practices can be adopted in any case; automated test drivers and continuous integration will help any software development project.

Obviously, applying all these practices as the literature suggests is sometimes difficult. Having a full-time on-site customer is sometimes impractical, for example, but having one part-time cus-

XP Resources on the Web

Much of the information on agile software processes and XP is available online. In the following, we list some major Web resources. (All sites were current as of 11 Dec. 2001.)

- Agile alliance manifesto • http://www.agilealliance.org.
- Agile modeling homepage • http://www.agilemodeling.com.
- Extreme Programming • http://www.extremeprogramming.org.
- Extreme Programming Roadmap • http://www.c2.com/cgi/wiki?ExtremeProgrammingRoadmap.
- Martin Fowler's articles • http://martinfowler.com/articles.html.
- XP Developer • http://www.xpdeveloper.com.
- Xprogramming.com • http://www.xprogramming.com.
- XP Universe conference program and papers: http://www.xpuniverse.com/xpuPapers.htm.
- XP 2000 conference program and papers • http://ciclamino.dibe.unige.it/xp2000.
- XP 2001 conference program and papers • http://www.xp2001.org/xp2001/conference/program.html.

tomer and good telecommunication can be sufficient to achieve the constant communication that makes or breaks an XP project.

Overall, XP is an agile software process that speeds up development and lets teams react flexibly to requirement changes, but some issues remain. Although there is much anecdotal evidence on its benefits, hard quantitative data is only sparsely available. Another issue is the scalability of the process: Aside from Kent Beck's recommendation of 5 to 15 people, we don't really know how big a team can get before XP breaks.

References

1. K. Beck, *Extreme Programming Explained: Embrace Change,* Addison Wesley Longman, Reading, Mass., 2000.
2. K. Beck and M. Fowler, *Planning Extreme Programming,* Addison Wesley Longman, Reading, Mass., 2001.
3. M. Fowler et al., *Refactoring: Improving the Design of Existing Code*, Addison Wesley Longman, Reading, Mass., 1999.
4. M. Beedle and K. Schwaber, *Agile Software Development with Scrum,* Prentice Hall, Upper Saddle River, N.J., 2001.
5. A. Cockburn, *Agile Software Development*, Addison Wesley Longman, Reading, Mass., 2001.
6. J. Highsmith, *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*, Dorset House, New York, 2000.
7. F. Maurer and S. Martel, "On the Productivity of Extreme Programming: An Industrial Case Study," draft paper; available at http://sern.ucalgary.ca/~milos/Library.htm (current 11 Dec. 2001).
8. J. Kerievsky, "Refactoring to Patterns," draft paper; available at http://industriallogic.com/papers/rtp110.pdf (current 11 Dec. 2001).
9. L. Williams et al., "Strengthening the Case for Pair Programming," *IEEE Software*, vol. 17, no. 4, July-Aug. 2000.
10. J. Zettel et al., "LIPE: A Lightweight Process for E-Business Startup Companies Based on Extreme Programming," *Proc. Third Int'l Conf. Product-Focused Software Process Improvement (PROFES 2001)*, Springer Verlag, Berlin, 2001.

Frank Maurer is an associate professor in computer science at the University of Calgary, Canada, where he is the head of the e-Business engineering group. His research interests include agile software processes, process support for virtual software corporations, and knowledge management. He is on the editorial boards of *IEEE Internet Computing* and *Electronic Transactions on AI Semantic Web*, and is a member of the IEEE Computer Society and the ACM. More information is available at http://sern.ucalgary.ca/~maurer.

Sebastien Martel is a graduate student in computer science at the University of Calgary, Canada. His research interests include agile software processes, process support for virtual software corporations, and artificial intelligence.

Readers can contact the authors at {maurer,smartel}@cpsc.ucalgary.ca.